```
class NQBacktracking:
    def __init__(self, x_, y_):

        self.ld = [0] * 30
        self.rd = [0] * 30
        self.cl = [0] * 30
        self.x = x_
        self.y = y_

    def printSolution(self, board):

        print("N Queen Backtracking Solution:\nGiven initial position of 1st queen at
row:",self.x,"column:",self.y,"\n",)
        for line in board:
            print(" ".join(map(str, line)))
    def solveNQUtil(self, board, col):
        if col >= N:
            return True
        if col == self.y:
            return self.solveNQUtil(board, col + 1)
        for i in range(N):
            if i == self.x:
                continue
            if (self.ld[i - col + N - 1] != 1 and self.rd[i + col] != 1) and self.cl[i] != 1:
                board[i][col] = 1
                self.ld[i - col + N - 1] = self.rd[i + col] = self.cl[i] = 1
                if self.solveNQUtil(board, col + 1):
                    return True
                board[i][col] = 0 # BACKTRACK
            self.ld[i - col + N - 1] = self.rd[i + col] = self.cl[i] = 0
        return False
    def solveNQ(self):
        board = [[0 for _ in range(N)] for _ in range(N)]
        board[self.x][self.y] = 1
        self.ld[self.x - self.y + N - 1] = self.rd[self.x + self.y] = self.cl[self.x] = 1
        if not self.solveNQUtil(board, 0):
            print("Solution does not exist")
            return False
        self.printSolution(board)
        return True
if __name__ == "__main__":
    N = 4
    x, y = 1, 3
    NQBt = NQBacktracking(x, y)
    NQBt.solveNQ()
```

OUTPUT:

```
N Queen Backtracking Solution:
Given initial position of 1st queen at row: 1 column: 3

0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0
```